O'REILLY®

Hadoop and Spark Performance for the Enterprise

Ensuring Quality of Service in Multi-Tenant Environments

Andy Oram

San Jose

London

Beijing

New York

Singapore

Hadoop and Spark Performance for the Enterprise

Ensuring Quality of Service in Multi-Tenant Environments

Andy Oram

Beijing · Boston · Farnham · Sebastopol · Tokyo

O'REILLY®

Hadoop and Spark Performance for the Enterprise
by Andy Oram

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Nicole Tache
Production Editor: Colleen Lobner
Copyeditor: Octal Publishing, Inc.
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

June 2016: First Edition

Revision History for the First Edition

2016-06-09: First Release
2016-07-15: Second Release

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-96319-7

[LSI]

Table of Contents

Hadoop and Spark Performance for the Enterprise: Ensuring Quality of Service in Multi-Tenant Environments

Modern Hadoop and Spark environments are busy places. Multiple applications being run by multiple users with wildly different workloads (HIVE queries, for instance, cheek-by-jowl with long MapReduce jobs) are contending for the same resources. And users are noticing the problems that result from contention: companies spend big bucks on hardware or on virtual machines (VMs) in the cloud, and don't get the results in the time they need.

Luckily, you can solve this without throwing in more and more money and overprovisioning hardware resources. Instead, you can aim for Quality of Service (QoS) in mixed workload, multitenant Hadoop and Spark environments. Throughout this report, I will use the term *distributed processing* to refer to modern Big Data analysis tools such as Hadoop, Spark, and HIVE. It's a very general term that covers long-running jobs such as MapReduce, fast-running in-memory Spark jobs that are often called "real-time," and other tools in the Hadoop universe.

Let's take a look at the waste left by distributed processing tasks. When developers submit a distributed processing job, they need to specify the amount of CPU required (by specifying the size of the system), the amount of memory to use, and other necessary parameters. But hardware requirements (CPU, network, memory, and so on) can change after the job is running. The performance company

Pepperdata, for instance, finds that a Hadoop job can sometimes go down to only 1 percent of its predefined peak resources. A research project named Quasar claims that “most workloads (70 percent) overestimate reservations by up to 10x, while many (20 percent) underestimate reservations by up to 5x.” The bottom line? Distributed systems running these jobs—whether on your own hardware or on virtual systems provisioned in the cloud—occupy twice as many resources as they actually need.

The current situation, in which developers lay out resources manually, is reminiscent of the segmented Intel architecture with which early MS-DOS programmers struggled. One has to go back some 30 years in computer history to find programmers specifying how much memory they need when scheduling a job. Most of us are fortunate enough to just throw a program onto the processor and let the operating system control its access to resources. Only now are distributed processing systems being enhanced with similar tools to save money and effort.

Virtually every enterprise and every researcher needs Big Data analysis, and they are contending with other people in their teams for resources. The emergence of real-time analysis—to accomplish such tasks as serving up appropriate content to website visitors, retail recommendations based on recent purchases, and so on—makes resource contention even more of an urgent problem. Now, you might not only be wasting money, you might miss a sale because a high-priority HBase query for your website was held up because an ad hoc MapReduce job monopolized disk I/O.

Not only are we wasting computer resources, we’re still not getting the timeliness we paid for. It is time to bring QoS to distributed processing. As described in the article “Quality of Service for Hadoop: It’s about time!,” the effort of QoS assurance would let programmers assign priorities to jobs, assured that the nodes running these jobs would give high-priority jobs the resources needed to finish within certain deadlines. QoS means that you can run distributed processing without constant supervision, and users (or administrators) can set priorities for different workloads, ensuring that critical jobs complete on time. In such a system, when certain Spark jobs have real-time requirements (for instance, to personalize web pages as they are created and delivered to viewers), QoS ensures that those jobs are given adequate response time. In a white paper, Mike Matchett, an analyst with Taneja Group, says:

> We think the biggest remaining obstacle today to wider success with big data is guaranteeing performance service levels for key applications that are all running within a consolidated…mixed tenant and workload platform.

In short, distributed processing environments need to evolve to accommodate the following:

- Multiple users contending for resources, as on operating systems
- Jobs that grow or shrink in hardware usage, sometimes straining at their resource limits and other times letting those resources go to waste
- Jobs of different priorities, some with soft real-time requirements that should allow them to override lower-priority or ad hoc jobs
- Performance guarantees, somewhat like Service Level Agreements (SLAs)

So let's see how these tools can move from the age of segmented computer architectures to the age of highly responsive scheduling and resource control.

Operating Systems, Data Warehouses, and Distributed Processing: A Common Theme

To get a glimpse of what distributed processing QoS could be, let's look at the mechanisms that operating systems and data warehouses have developed over the years.

Operating systems make it possible for multiple users running multiple programs to coexist on a relatively small CPU with access to limited memory. Typically, a program is assigned a specific amount of CPU time (a *quantum*) when it starts and is forced to yield the processor to another when the time elapses. Different processes can be started with higher priorities to get more time or lower priorities to get less time. When the process regains control of the processor, the operating system scheduler might assign it the same time quantum, or it might reward or punish the process by changing the quantum or its priority.

For instance, the current Linux scheduler rewards a process that yields the CPU before using up its assigned quantum; this usually occurs because the process needs to read or write data to disk, the network, or some other device. Such processes are assigned a higher priority and therefore are chosen more quickly to run again. This cleverly solves a common problem: treating batch processes that run background tasks differently from interactive processes that ought to respond as quickly as possible to a user's mouse click, keystroke, or swipe.

Here's how it works: interactive processes wait frequently for user activity, so they usually yield the processor quickly before using much of their quanta. Because the scheduler raises their priority, they are less likely to wait for other processes before starting up when the user presses a button or key. I/O-bound processes are not always interactive, and an interactive process can sometimes be CPU-bound (for instance, if it has to render a complex graphic) but the correspondence holds well enough to make most people feel that their programs are responding quickly to input.

However, the programmer is not at the mercy of the scheduler to determine a process's priority. In addition to assigning a priority manually, the programmer can (on most operating systems) designate a process as real-time or first-in-first-out (FIFO). Such processes preempt all non-real-time processes and therefore have a high likelihood of meeting the programmer's goal, whether it's an immediate response to input (think of a car braking when the user presses the brake pedal) or just finishing as fast as possible (think of a web server deciding what ad to serve on the page). The latter kind of speed is comparable to what many data analysts need when running Spark jobs.

Another aspect of QoS is less relevant to this report: locality. A scheduler will try to run each process on the same CPU where it ran before, so long as there is not a big disparity in loads on different CPUs. But when one CPU is very heavily loaded and another is routinely idle, the scheduler will move a process. This has a performance cost because memory caches must be cleared and reloaded. The corresponding issue in batch-data jobs is to keep processes that use the same data (such as a map and a reduce) on the same node in the network. Here, distributed processing tools such as Hadoop are quite intelligent, minimizing moves that would require large amounts of data to be copied or reloaded.

Operating systems offer programmers another important service: they report statistics about the use of CPU, memory, and I/O. Examples of this are Task Manager in Windows or the *top*, *iostat*, and *netstat* commands in Linux. This lets programmers troubleshoot a slow system and make necessary changes to processes.

It should be noted, finally, that operating system schedulers have limitations, particularly when it comes to ordering I/O. It is usually the job of the disk controller, a separate special-purpose CPU, to arrange reads and writes as efficiently as possible. Unfortunately, the disk controller has no concept of a process, doesn't know which process issued each read or write, and can't take operating system priorities into account. Therefore, a high-priority process can suffer priority inversion—that is, lose out to a lower-priority process—when performing I/O.

Data warehouses have also developed increasingly sophisticated and automated tools for capacity planning, data partitioning, and other performance management tasks. Because they deal with isolated queries instead of continuous jobs, their needs are different and focus on query optimization.

For instance, Teradata provides resource control and automated request performance management. It runs disk utilities such as defragmentation and automatic background cylinder packing (AutoCylPack), a kind of garbage collection for space that can be freed. Oracle, in addition to memory management, uses data from its Automatic Workload Repository to automatically detect problems with CPU usage, I/O, and so on. In addition to detecting resource-hogging queries and suggesting ways to tune them, the system can detect and solve some problems automatically without a restart.

In summary, we would like distributed processing like Hadoop to behave more like operating systems and data warehouses in the following ways:

- Understanding different priorities for different jobs
- Monitoring the resource usage of jobs on an ongoing basis to see whether this usage is rising or falling
- Rob low-priority jobs of CPU, memory, disk I/O time, and network I/O (while trying to minimize impacts on them) when it's necessary to let a high-priority job finish quickly

- Raise and lower the resource limits imposed by the jobs' containers to reflect the jobs' resource needs and thus meet the previous goal of promoting high-priority jobs
- Log resource usage, recording when a change to container limits was required, and display this information for future use by programmers and administrators

Now we can turn to distributed systems, explore why they have variable resources needs, and look at some solutions that improve performance.

Performance Variation in Distributed Processing

Hadoop and Spark jobs are launched, usually through YARN, with fixed resource limits. When organizations use in-house virtualization or a cloud provider, a job is launched inside a VM with specified resources. For instance, Microsoft Azure allows the user to specify the processor speed, the number of cores, the memory, and the available disk size for each job. Amazon Web Services also offers a variety of instance types (e.g., general purpose, compute optimized, memory optimized).

Hadoop uses *cgroups*, a Linux feature for isolating groups of processes and setting resource limits. cgroups can theoretically change some resources dynamically during a run, but are not used for that purpose by Hadoop or Spark. cgroups' control over disk and network I/O resources is limited.

But as explained earlier, the resource needs of distributed processing can actually swing widely, just like operating system processes. There are various reasons for these shifts in resource needs.

First, an organization multitasks. In an attempt to reduce costs, it schedules multiple jobs on a physical or virtual system. Under favorable conditions, all jobs can run in a reasonable time and maximize the use of physical resources. But if two jobs spike in resource usage at the same time, one or both can suffer. The host system cannot determine that one has a higher priority and give it more resources.

Second, each type of job has reasons for spiking or, in contrast, drastically reducing its use of resources. HBase, for instance, suffers resource swings for the same reasons as other databases. It might have a period of no queries, followed by a period of many simultaneous queries. A query might transfer just one record or millions of records. It might require a search through huge numbers of records—taking up disk I/O, network I/O, and CPU time—or be able to consult an index to bypass most of these burdens. And HBase can launch background tasks (such as compacting) when other jobs happen to be spiking, as well.

MapReduce jobs are unaffected by outside queries but switch frequently between CPU-intensive and I/O-intensive tasks for their own reasons. At the beginning, a map job opens files from the local disk or via HDFS and does seeks on disk to locate data. It then reads large quantities of data. The strain on I/O is then replaced by a strain on computing to perform the map calculations. During calculations, it performs I/O in bursts by writing intermediate output to disk. It might then send data over the network to the reducers. The same kinds of resource swings occur for reduce tasks and for Spark. Each phase can use seconds or minutes.

Figure 1-1 shows seven of the many statistics tracked by Pepperdata. Although Pepperdata tracks hardware usage for every individual process (container or task) associated with each job and user, the charts in Figure 1-1 are filtered to display metrics for a particular job, with one line (red) showing the usage for all mappers added together and another line (green) for all reducers added together. Each type of hardware undergoes vertiginous spikes and drops over a typical run.

All this is recorded at the operating-system level, as explained earlier. But Hadoop and Spark jobs don't monitor those statistics. Most programmers don't realize that these changes in resource use are taking place. They do sometimes use popular monitoring tools such as Ganglia or Hadoop-specific tools to view the load on their systems, and such information could help programmers adjust resource usage on future jobs. But you can't use these tools *during* a run to change the resources that a system allocates to each job.

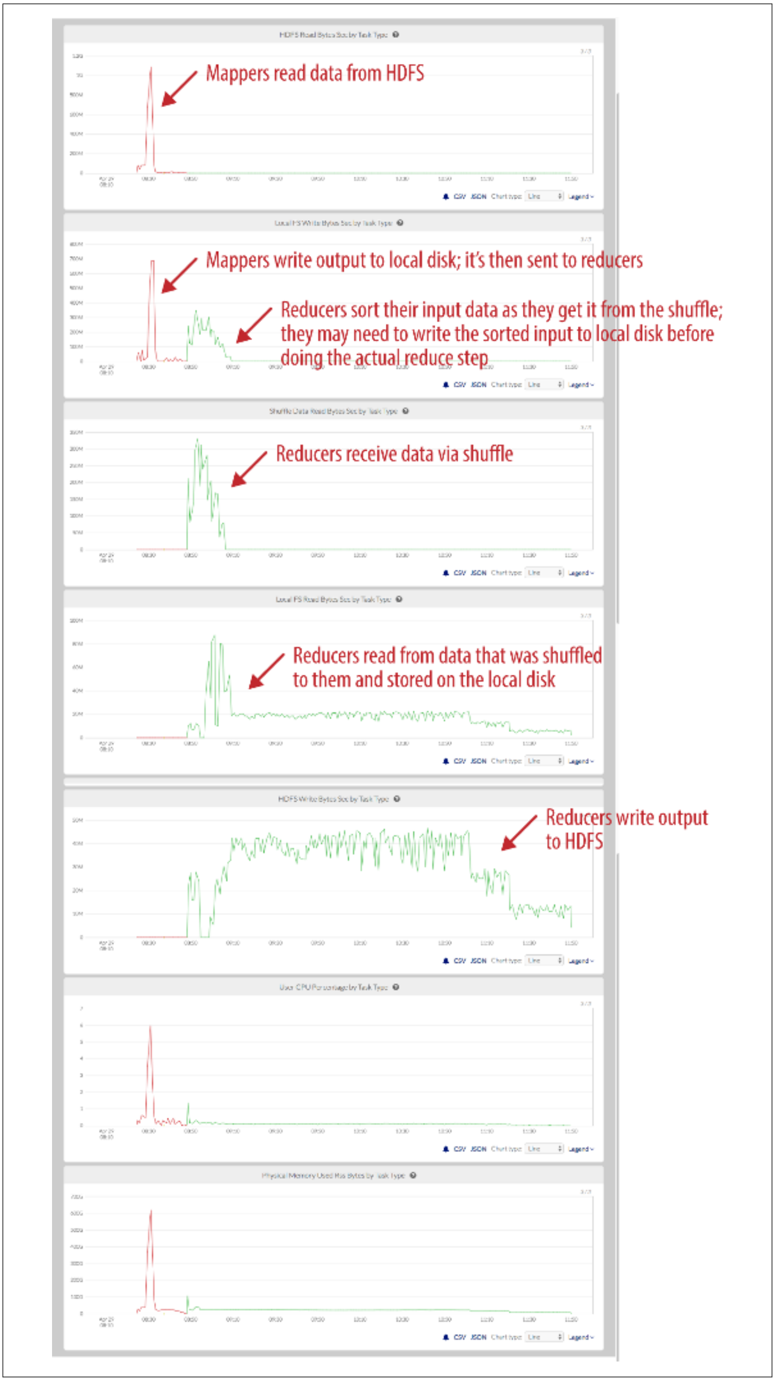

Figure 1-1. Spikes and drops during a normal MapReduce run

One posting suggests adding 25 percent to each resource allocation to account for spikes. This is a crude workaround for inevitable contention and is not likely to prevent it when applications get hot. Adding more resources is like putting more wheels on a car—it doesn't make it run faster.

Finally, it should be noted that YARN often deals with slowdowns by duplicating tasks. It allows each task to proceed until one finishes, and then kills off the duplicate tasks. Although this enhances robustness by ensuring that each task is completed, it does so by using more resources and therefore placing a still greater burden on systems that were slow to begin with.

As we mentioned earlier, resources are specified when the job launches and cannot be changed during the job's execution. There is no overarching supervisor looking at all the jobs and the resources on the system. Somewhat intrusive changes to the system are required if we want to achieve the same flexibility that operating systems offer to their users.

Improving Distributed Processing Performance

Two systems have been unveiled so far to remove waste and improve the performance of distributed processing; that is, to change the resources allocated to a job *in the middle of its run*.

Quasar (mentioned at the beginning of this report) is an academic project that dynamically changes resource allocations to jobs running in clouds, Hadoop and Spark jobs, and various distributed database systems. Insights from this project are being incorporated into Mesos, an open source scheduling system.

Quasar bases its changes on profiles that it creates in advance. These profiles come from running a number of typical jobs. When deployed on a production system, Quasar runs each new job for a few seconds (10 to 60 seconds, according to a video presentation) and assigns the job to a profile, guessing that the job will act like known jobs in the past. But if the new job acts differently, Quasar can also switch it to a different profile during its run.

Following the chosen profile, Quasar changes the resource allocations on a minute-by-minute basis for the job. (The research paper doesn't explain how they make the changes.) The paper has found, in one set of benchmarks, a 47 percent improvement in overall performance.

Another system that dynamically alters resource requirements for distributed processing is Pepperdata. It responds to actual changes in resource usage instead of working from prestored profiles like Quasar. Pepperdata's monitoring and potential resource changes take place once per second instead of once per minute. And although Quasar notes whether a profile is sensitive to contention from other jobs, Pepperdata can actually take that contention into account.

This is because Pepperdata runs a supervisor on each physical or virtual node, monitoring it for overall resource usage. Pepperdata also lets programmers assign priorities to jobs. Thus, a high-priority job that might fly along fine on its own but that suffers because it happens to run when other jobs have high resource usage, can be rescued by Pepperdata's supervisor.

A sample case study of optimization is provided by the company Upsight, which measures key performance indicators for business, performs user segmentation for marketing, tailors mobile messaging, and does similar data-intensive tasks. The company deployed Pepperdata to guarantee HBase performance. The criterion for success was allowing HBase to achieve stable HDFS write throughput, regardless of other MapReduce activity on the cluster.

The top chart in Figure 1-2 shows a number of processes contending for a processor during a typical period of time. At the bottom is a purple line representing HBase, which was the time-critical process requiring QoS support in this example. HBase runs fairly evenly for the first hour. Then, it varies widely in processor use and even disappears as it is held up by various MapReduce tasks that come and go.

The bottom chart in Figure 1-2 shows the same load running under the control of Pepperdata's QoS. You now see a fairly even line above the purple bar that represents the time-critical HBase process. By stealing a tiny, barely noticeable percentage of resources from other processes, Pepperdata gives HBase what it needs to reliably serve its users.

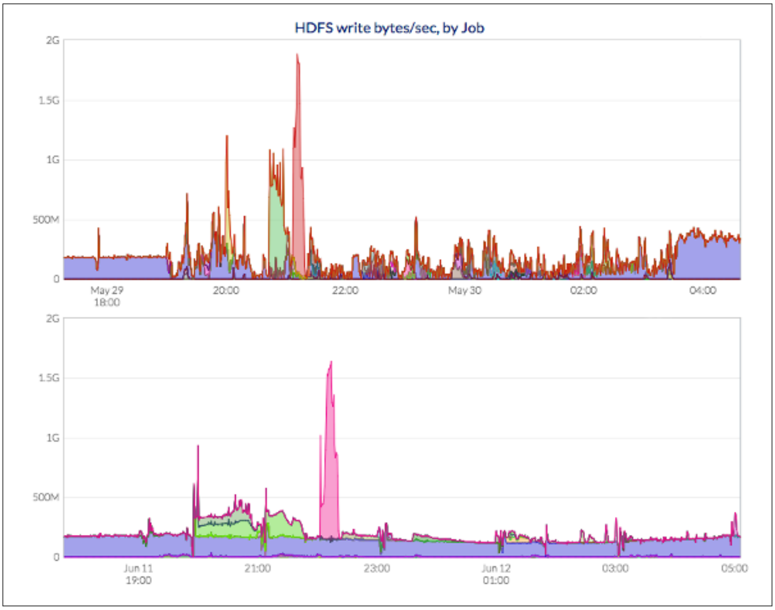

Figure 1-2. Evening out a critical application

Pepperdata monitors a number of resources individually. In Figure 1-3, a brief but very sharp spike shows that a large number of sockets were opened on a host simultaneously. This happened to occur because several jobs reached a point in their work where they needed to open network connections; spreading these out a bit would improve performance.

Figure 1-4 shows a Pepperdata display uncovering a waste of resources that might be a surprise. The leftmost chart shows local reads carried out by a variety of jobs; the numbers are quite low. The central chart shows reads on other nodes on the rack, whereas the rightmost chart shows remote reads. Both of those spike much higher. Ideally, one would want to keep reads as local as possible, so this chart can alert the programmer that the layout of data in these jobs is suboptimal.

Pepperdata delves into operating system resources that programmers routinely use to check performance, but that YARN and other job schedules don't access. Pepperdata uses whatever facilities are available in the Java Virtual Machine (JVM) and operating system to alter the resources allocated to a job. For instance, currently they instruct the JVM running a container to make a running thread sleep. They can also work on the Linux kernel level to raise or lower tasks' priorities. Other mechanisms might be used in the future as they become available.

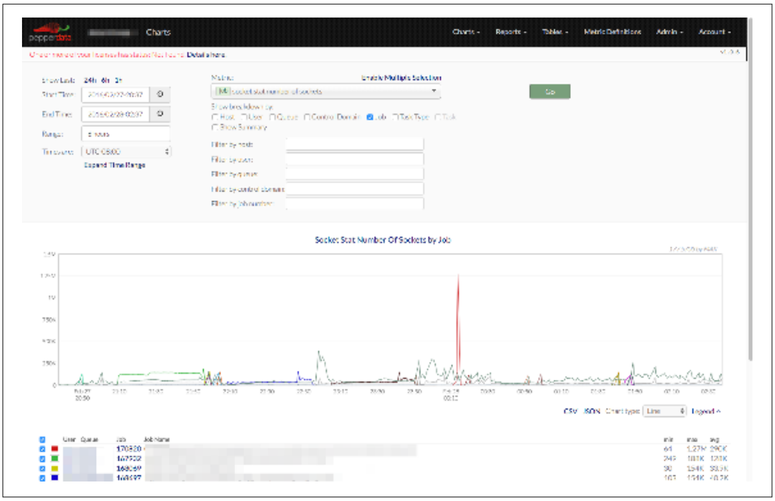

Figure 1-3. A spike in network requests

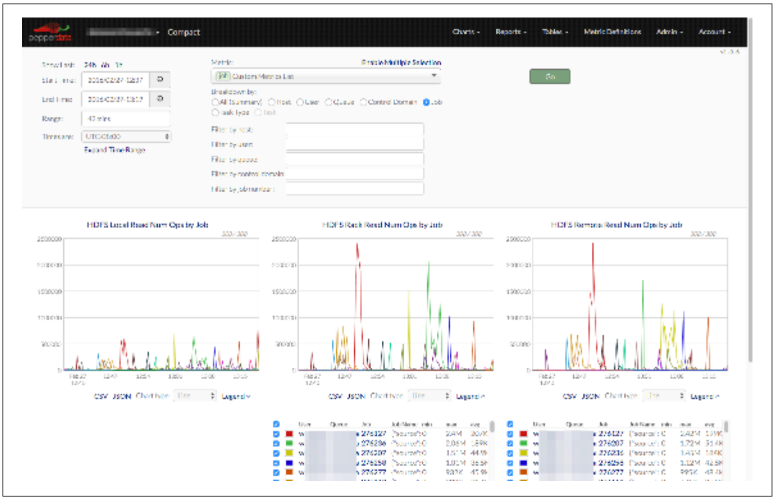

Figure 1-4. Remote requests outweigh local requests

Pepperdata can also inform YARN that hosts have free resources when YARN doesn't previously know of them. As explained earlier, YARN assigns the maximum number of resources requested by the programmer when it runs each job on a host. When resources drop and a lot of free CPU, memory, and I/O become available, YARN does not normally know, but Pepperdata does. When YARN is provided that information, it can schedule new processes for existing jobs, or even entirely new jobs.

Pepperdata reports very low overhead for monitoring: at most 0.1 percent of the total CPU power of the node. When a higher-priority job takes resources from lower-priority jobs, it usually lasts for only a few seconds, 2 or 3 percent of the lower-priority job's run. So, time requirements can be met with negligible effects on the low-priority jobs. Overall improvements in capacity or throughput are in the range of 30 to 50 percent.

Finally, Pepperdata supervisors report statistics to a central system where programmers can check statistics, see when changes to resources were automatically made, and set custom alerts.

Conclusion

Distributed processing faces complex performance issues. Tools provided by operating systems to *monitor* resources are not always matched by tools to *adjust* those resources—and things become more difficult as you move to up the stack to virtual containers.

Distributed processing jobs vary widely in resource usage, and the ample size of these jobs makes control over resources a critical capability for organizations that depend on data analysis to meet their business goals. Tools to address performance problems in these technologies were critical to their mainstream adoption and support by IT operations teams. They were able to support a much broader variety of use cases more reliably. For distributed processing like Hadoop and Spark to contribute their march toward mainstream, it needs to develop capabilities similar to operating systems and data warehouses.

Quasar and Pepperdata have shown that substantial improvements can be earned through dynamic load control, and capabilities will probably get better as the field learns more about what is needed and how to integrate the different levels of performance.

About the Author

Andy Oram is an editor at O'Reilly Media. An employee of the company since 1992, Andy currently specializes in programming topics. His work for O'Reilly includes the first books ever published commercially in the United States on Linux, and the 2001 title *Peer-to-Peer*.